Runtime Monitoring for Executable DSLs

Dorian Leroy, Pierre Jeanjean, Erwan Bousse, Manuel Wimmer, Benoit Combemale

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Runtime Monitoring for Executable DSLs

Dorian Leroy[a] Pierre Jeanjean[b] Erwan Bousse[c]
Manuel Wimmer[a] Benoit Combemale[bd]

a. JKU Linz, Austria
b. Inria, Univ. Rennes, CNRS, IRISA, France
c. Univ. Nantes, LS2N, France
d. Univ. Toulouse, IRIT, France

Abstract Runtime monitoring is a fundamental technique used throughout the lifecycle of a system for many purposes, such as debugging, testing, or live analytics. While runtime monitoring for general purpose programming languages has seen a great amount of research, developing such complex facilities for any executable Domain Specific Language (DSL) remains a challenging, reoccurring and error prone task. A generic solution must both support a wide range of executable DSLs (xDSLs) and induce as little execution time overhead as possible. Our contribution is a fully generic approach based on a temporal property language with a semantics tailored for runtime verification. Properties can be compiled to efficient runtime monitors that can be attached to any kind of executable discrete event model within an integrated development environment. Efficiency is bolstered using a novel combination of structural model queries and complex event processing. Our evaluation on 3 xDSLs shows that the approach is applicable with an execution time overhead of 121% (on executions shorter than 1s), to 79% (on executions shorter than 20s) making it suitable for model testing and debugging.

Keywords runtime monitoring, temporal property language, executable modeling, domain-specific languages

Keywords Executable DSLs, Runtime Monitoring

1 Introduction

A large amount of Domain-Specific Languages (DSLs) are used to represent behavioral aspects of systems in the form of *behavioral models* (*e.g.,* [BCCG07, HLN+90, OAS07, Obj13b]). To enable the dynamic analysis of such models, a lot of efforts have been

Dorian Leroy, Pierre Jeanjean, Erwan Bousse, Manuel Wimmer, Benoit Combemale. *Runtime Monitoring for Executable DSLs.* In Journal of Object Technology, vol. 19, no. 2, 2020, pages 6:1–23. doi:10.5381/jot.2020.19.2.a6

made to facilitate the design of so-called *xDSLs* (*e.g.,* [CCP12, EHHS00, MLWK13, TCGT14]), which enable the execution of conforming models. Two approaches are commonly used to define the execution semantics of xDSLs, namely operational semantics (*i.e.,* interpretation) and translational semantics (*i.e.,* compilation). While we consider both kinds of semantics in this paper [1] , we focus our work on *discrete-event* semantics.

When designing or using executable models, a wide range of cases require observing and analyzing the execution of models at *runtime*, such as the detection of breakpoints when debugging, the evaluation of oracles when testing, or controlling the health of a running system (*i.e.,* live analytics). Producing an analysis while observing an execution is achieved by what is commonly called *runtime monitoring*, a fundamental technique that ascertains whether a running system fulfills or violates temporal properties. Providing support for runtime monitoring for xDSLs is therefore of great importance in order to face the aforementioned situations.

While the topic of runtime monitoring for general purpose languages has seen a sizable amount of research [CFAI17], providing such facilities for any existing or new xDSL remains a challenging task. First, xDSLs come in many shapes and forms. Thus, for each new xDSL, corresponding tooling must be developed or adapted from existing tooling, which is tedious and error-prone. Second, a general challenge of runtime monitoring is to maintain a low execution time overhead.

To face these challenges we propose an approach to create runtime monitors that can be attached to models conforming to any xDSL, in the context of their execution in an integrated development environment. We define a temporal property language where temporal aspects are based on Property Specification Patterns (PSPs) by Dwyer *et al.* [DAC98], and where structural aspects are based on the Viatra Query Language (VQL) model querying language [BURV11]. We provide a compiler for our language that adapts the semantics of the PSPs to the particularities of runtime monitoring, thereby compiling properties into efficient synchronous runtime monitors based on Complex Event Processing (CEP). Finally, we introduce a property manager that both integrates at runtime the structural and temporal concerns of our runtime monitors, and enables direct use of the monitors with xDSLs considered in our scope.

We provide an implementation of the proposed approach as part of the GEMOC Studio [BDV$^+$16], an Eclipse-based language and modeling workbench for xDSLs. The monitors compiled from the temporal properties are executed using Esper[2] for temporal aspects and VQL [BDH$^+$15] for structural aspects.

We perform a quantitative evaluation of the overhead induced by the approach with regard to the kind of property monitored, the size of the models, the footprint of the properties (*i.e.,* the number of observed model elements) and the base execution time of the models, on 3 xDSLs. We show that the approach is applicable with an average execution time overhead ranging from 121% (for very short execution times, inferior to 1s) to 79% (for longer execution times, inferior to 20s). We conclude that the approach is well-suited for model testing and debugging, where such an average execution time overhead is reasonable.

The remainder of this paper is structured as follows. Section 2 presents the background and the motivation for this work. Section 3 provides an overview of our contributions. Section 4 presents the temporal property language. Section 5 presents

[1] More precisely, we only consider translational semantics that include a mapping back in the source model (*i.e.,* a simulation of the execution back in the source model, while the real execution happens in the model or code produced by the compiler).

[2] `http://www.espertech.com/esper/`

its execution semantics. Section 6 explains the implementation of the approach in the GEMOC Studio. Section 7 presents the evaluation. Section 8 discusses related work. Finally, a conclusion and future research directions are presented in Section 9.

2 Background and Motivation

In this section, we first scope the xDSLs considered in our approach, *i.e.,* xDSLs defined by a metamodel-based abstract syntax and a discrete-event execution semantics. We then briefly introduce two core foundations of the proposed approach: runtime monitoring and complex event processing. Finally, we motivate the approach using an illustrative example.

2.1 Background

Executable DSLs. An xDSL is composed of both an abstract syntax defining the concepts of the considered domain, and an execution semantics defining the meaning of these concepts[3]. In this paper, we focus on DSLs where (1) the abstract syntax is provided as a *metamodel* defined using an object-oriented metamodeling language (*e.g.,* MOF [Obj16] or Ecore [SBPM08]), and (2) the execution semantics is provided as an operational semantics (*i.e.,* an interpreter) or a translational semantics (*i.e.,* a compiler). In both cases, we consider that the semantics contains a data structure representing the (domain-specific) *model state* during the execution. In the case of an operational semantics, the model state is *directly modified* by the execution rules of the semantics. In the case of a translational semantics, the model state is *reconstructed* from the actual state of the execution—*i.e.,* the state of the executed model or code produced by the compiler.

More precisely, we consider the model state to be defined in an *execution metamodel* that extends the abstract syntax metamodel using a non-intrusive extension mechanism, such as *package merge* [Obj13a] or *aspect weaving* [JCB+13]. Then, an in-place endogenous transformation is performed by the semantics on this model state, either on its own for operational semantics or as a result of receiving feedback on the actual execution for translational semantics. This transformation results in the execution of the model.

While this transformation may result in a broad range of reachable model states, we consider that only a subset of these states are *observable*, the rest being temporary, inconsistent states. For instance, an execution semantics may define that the state is only observable in between the application of execution rules. It is thus up to the execution semantics to both allow components to register as observers of the execution, and to notify observing components of either the occurrence of observable states or the applications of atomic rules.

Figure 1 shows an example of an Activity Diagram DSL with an operational semantics. This example is a simplified version of the part of fUML [Obj13b] related to the control flow of activities. At the top left, the abstract syntax defines an Activity as a set of inter-connected Node and Edge objects, with several types of nodes. InitialNode and FinalNode mark the beginning and the end of the Activity. A ForkNode starts concurrent execution branches, which can be joined back in a JoinNode. An Action represents an opaque action realized in the process. At the top right, the execution metamodel defines what is the model state of an activity by introducing a

[3]As our approach is agnostic to concrete syntax, we do not consider it in this paper

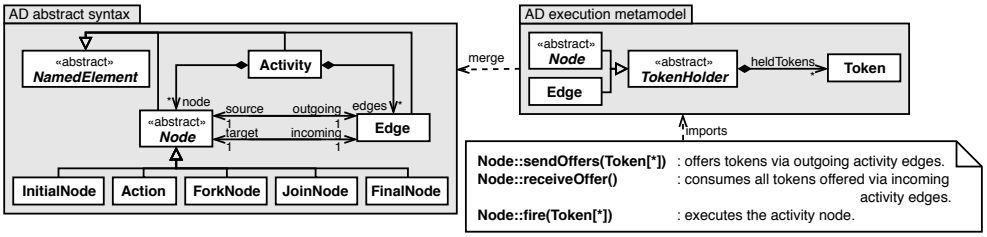


Figure 1 – Activity Diagram DSL with an operational semantics.

new metaclass called TokenHolder. This metaclass enables Node and Edge elements to contain Token elements. Lastly, the operational semantics of the DSL is defined by a set of execution rules, and is broadly based on a token flow starting in the initial node and ending in the final node. Three execution rules for the Node metaclass are shown at the bottom: *receiveOffers* and *sendOffers* that respectively take and put tokens from edges, and *fire* that (1) triggers *receiveOffers*, (2) performs the opaque action of the node, (3) triggers *sendOffers*.

Runtime Monitoring. Runtime monitoring consists in observing the internal state of a system during its execution to check whether the system satisfies or violates a correctness specification—usually provided as a temporal property [LS09]—on this particular execution. In this paper, we focus on a refined category of online monitoring called *synchronous monitoring* [CFAI17]. With synchronous monitoring, the runtime monitor is tightly coupled with the system. More precisely, the execution of the system is suspended whenever the monitor checks whether a received event violates the property or not. Compared to asynchronous monitoring, this allows timely (as opposed to late) detection of property violation or satisfaction, at the cost of performance decrease.

In [BSVV18, BSVV19], Búr *et al.* successfully used the VIATRA Query Language (VQL) for runtime monitoring of distributed safety properties over models@runtime. VQL is a declarative graph query language which allows to directly express queries using the domain-specific concepts expressed in metamodels. Its successful use emphasizes its relevance and motivates our use of this technology to realize our approach.

Complex Event Processing. The goal of CEP is to identify meaningful events over streams of simpler events with queries on both the data carried by the events and the before and after relationships between them. Essentially, CEP systems allow to perform temporal pattern matching over streams of events and produce a new stream of complex events as a result. An example of well-known CEP framework is Esper, which is an open-source Java-based system that provides a DSL for event processing called Event Processing Language (EPL). This DSL allows to write queries—called EPL statements—that continuously analyze events within a stream to detect situations of interest and produce a new stream of events containing properties selected from the matching events. Java objects can then subscribe to this new event stream to be notified each time an event is inserted into the stream.

2.2 Motivational Example

Figure 2 shows an example Activity conforming to the Activity Diagram DSL shown in Figure 1. The shown activity is a simplified process of withdrawing cash at an

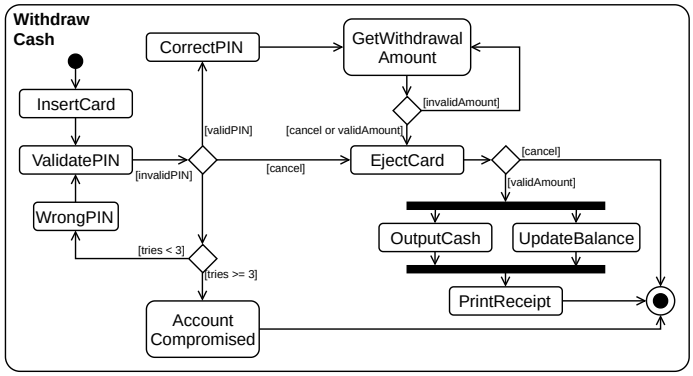


Figure 2 – The *Withdraw Cash* Activity of an ATM.

ATM. First, the card is inserted into the ATM. Then, the PIN entered by the user is checked. If invalid, a new PIN is requested and checked, until the correct PIN is entered, the number of failed attempts reaches 3, or the user cancels the process. If no successful attempt is made in 3 tries, the card is swallowed and the account flagged as compromised. On a successful attempt, the user is asked for a withdrawal amount until she or he enters a valid amount (with regard to her or his account balance, the amount being a multiple of 10, etc.). Alternatively, the user can choose to cancel her or his withdrawal. Should the user choose to cancel the process at the PIN validation stage or at the withdrawing stage, the card is ejected and the activity ends. However, if a valid amount has been entered, the ATM first ejects the card, then concurrently outputs the correct amount of cash and updates the account balance of the user. Finally, the ATM prints a receipt indicating the amount withdrawn and the new account balance.

There are several temporal properties one might want to monitor on such an activity. For instance, it would be interesting to monitor whether the card always ends up being ejected once the correct PIN has been entered (`P1`). Another interesting property that can be monitored is that the card is always ejected before cash is distributed, to prevent people from forgetting their card in the ATM (`P2`). A last property one can monitor is that an invalid PIN is not entered more than two times between the moment a card is inserted and a correct PIN is entered (`P3`). Monitoring these properties can be achieved by instrumenting the execution semantics of the Activity Diagram DSL. However, it requires to design an instrumentation technique that is tightly coupled to the Activity Diagram language and the metalanguage used to write its execution semantics. This hampers reusing this instrumentation technique, requiring an adaptation for each new xDSL or metalanguage to support.

In Section 3 we provide an overview of the proposed approach, before detailing its inner workings in Sections 4 and 5.

3 Approach Overview

In this section, we provide an overview of the approach from the modeler's point of view, both at design time and at runtime. This overview is illustrated by Figure 3.

Design Time At design time, modelers define temporal properties using the *temporal property language* proposed as part of the approach, as shown at the top of Figure 3. Modelers first specify what are the states of interest to detect in the execution, and

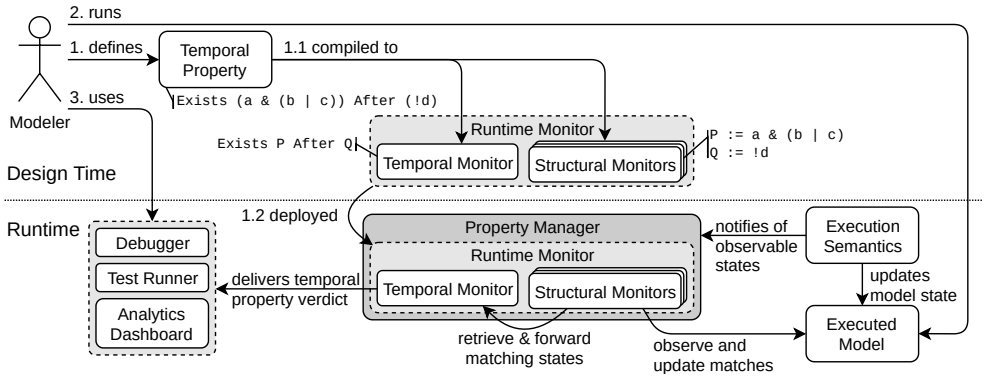


Figure 3 – Overview of the approach.

then define the temporal pattern that must be observed with these states during the execution for the property to be satisfied. For the definition of the states of interest, the approach relies on structural patterns written in VQL [BURV11] by the modeler using domain concepts, thereby allowing her to leverage her domain knowledge. For the definition of the temporal pattern, the approach relies on the PSPs, which makes it easy to express the most-used kinds of temporal properties [BGPS12]. The resulting abstract syntax of the proposed property language is presented in Section 4.1.

The compiler will then process a temporal property by separating the structural concern of a property from its temporal concern, and respectively produce *structural monitors* that are each able to detect when a specific structural pattern is encountered, and a CEP-based *temporal monitor* able to reason over time on states detected by structural monitors. An overview of the execution semantics of the proposed property language is presented in Section 4.2, and a focus on the translation scheme used to derive runtime monitors from temporal patterns is given in Section 5.

Runtime At runtime, the runtime monitors derived from temporal properties are deployed into a property manager, as illustrated at the bottom of Figure 3. This property manager handles the connection between the structural and temporal monitors, the execution semantics, the running model and the various property listeners (*e.g.,* debugger, test runner, analytics dashboard, etc.).

At the start of the execution, the structural monitors are configured to update their matches whenever changes occur in the state of the running model, as shown on the lower left part of Figure 3. These matches are however only retrieved when an observable model state is reached, of which the property manager is notified by the execution semantics of the DSL, as earlier explained in Section 2.1. This is to avoid evaluating the property on inconsistent model states.

Pattern matches are then sent to the temporal monitor, as shown on the lower middle part of Figure 3. The temporal monitors take these new matches (or absence thereof) into consideration to evaluate whether the property is satisfied or violated, or if no verdict can be rendered yet. Then, in case a final verdict is rendered, it is delivered to the property manager, which relays it by notifying the potential property listeners, as shown on the right lower part of Figure 3.

How structural and temporal monitors work together with the property manager is described in Section 4.2.

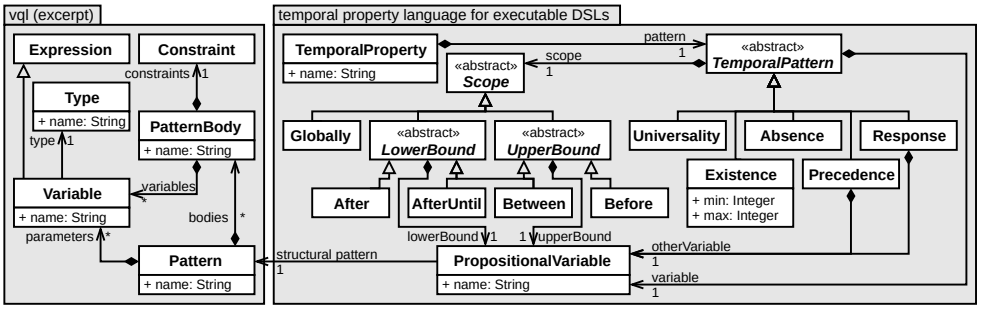


Figure 4 – Abstract syntax of the proposed temporal property language.

4 Temporal Property Language for xDSLs

In this section, we present our temporal property language for xDSLs, and how this language can be used for the runtime monitoring of executable models. The proposed language is based on VQL for expressing structural patterns, and on the PSPs for expressing temporal patterns. We first define the abstract syntax of the language, and then give an overview of the translation scheme used to derive runtime monitor from properties defined with the language.

4.1 Abstract Syntax

The abstract syntax of the property language is shown as a metamodel on Figure 4, and is presented thereafter.

Propositional variables and structural patterns. A temporal property language must be able to describe *propositional variables* that each represents some truth about the state of the executed model. Such a variable may equal either *true* or *false* for a given state of the model, *i.e.,* it must behave as a predicate taking the state of the model as parameter. In this paper, we choose to define such predicates using *structural patterns*. A structural pattern is a partial description of a model that returns *true* if a given model—here, the state of the executed model—matches what it describes, and *false* otherwise. Instead of redefining a structural patterns description language, our temporal property language relies on the existing VQL model querying language. In Figure 4, this is represented by the PropositionalVariable metaclass, which has a `name` and an associated VQL Pattern. Such Pattern elements can be defined using the domain concepts of any DSL whose metamodel is imported, by using its metaclasses as Type elements. Due to space limitations, we do not present further VQL in this paper, and we refer the reader either to examples shown at the end of the section, or to the paper presenting VQL [BURV11].

Scopes. Next, a temporal property language must be able to specify the *scope* of a given property. A scope determines a segment of the execution where a temporal pattern (see below) is expected to be satisfied. On each scope activation, the temporal pattern of the property must be observed, otherwise the property is violated. Instead of reinventing such concept, we adapt the different kinds of scopes used for the PSPs.

In Figure 4, this is represented by the Scope abstract metaclass, which is subdivided in different subclasses. The Global scope is active during the whole execution. The Before scope is active until its `upperBound` propositional variable becomes `true`.

```
import "http://org.tetrabox.activitydiagram/ad/"

pattern activeNode(nodeName : java.lang.String) {
  ActivityNode.name(node, nodeName);
  check(node.heldTokens.empty() = false);
}

pattern InsertCard() {activeNode("InsertCard");}

pattern CorrectPIN() {activeNode("CorrectPIN");}

pattern WrongPIN() {activeNode("WrongPIN");}

pattern OutputCash() {activeNode("OutputCash");}

pattern EjectCard() {activeNode("EjectCard");}
```

Figure 5 – Structural patterns for Figure 2.

P1	`exists 1 EjectCard after CorrectPIN`
P2	`EjectCard precedes OutputCash globally`
P3	`exists [0,2] WrongPIN between InsertCard and CorrectPIN`

Table 1 – Example properties for Figure 2.

Conversely, the After scope is active after its `lowerBound` propositional variable becomes `true`, and until the end of the execution. The AfterUntil scope becomes active after its `lowerBound` propositional variable becomes `true` until the end of the execution or its `upperBound` propositional variable becomes `true`, whichever comes first. Finally, the Between works similarly, except that a scope becomes *potentially active* after the `lowerBound` propositional variable becomes `true` and will retro-actively become *active* only once its `upperBound` propositional variable becomes `true`. This means that a verdict cannot be rendered before the scope is confirmed to have been active, and that, as opposed to the AfterUntil scope, the end of the execution is not a valid upper bound for a Between scope.

Temporal patterns. Finally, a temporal property language must be able to specify *temporal patterns*, *i.e.,* how propositional variables should take their values when the scope of the corresponding temporal property is active. Like scopes, instead of reinventing temporal patterns, we reuse and adapt the PSPs.

In Figure 4, this is represented by the TemporalPattern abstract metaclass, which is subdivided in different subclasses. The Universality (resp. Absence) pattern indicates that its associated propositional variable must be and remain `true` (resp. `false`) during each active scope of the property. The Existence pattern indicates its associated propositional variable must be `true` on a number of execution states that is within the lower and upper bounds of the pattern represented by its `min` and `max` attributes. The Precedence pattern indicates that its associated propositional variable must become `true` before another does during each active scope. The Response pattern indicates that during each active scope, when its associated propositional variable becomes `true`, its other propositional variable eventually becomes `true` as well. While the PSPs feature two more patterns, namely the *chained response* and *chained precedence* patterns, we leave them for future work as they are among the least used patterns.

Example Properties. To define the temporal properties introduced in Section 2.2, we first need to define the propositional variables corresponding to when specific activity nodes are reached, specifically the `CorrectPIN`, `WrongPIN`, `EjectCard` and `OutputCash` activity nodes. This can be done by defining a general, parameterized structural pattern looking for an ActivityNode whose name is passed as a parameter and whose `heldTokens` property contains at least one Token, as shown on Figure 5. Each propositional variable can then be defined using this general structural pattern.

Once the propositional variables are defined, the temporal properties can be written

as shown on Table 1. `P1` is defined as an Existence pattern over an After scope. `P2` is expressed through a Precedence pattern over a Global scope. Finally, `P3` is written as an Existence property with an upper bound, over a Between scope.

4.2 Overview of the Translation Scheme

We provide a translational semantics for the proposed temporal property language consisting in separately compiling the structural and temporal patterns of properties into structural and temporal runtime monitors. The resulting monitors are then integrated using a property manager.

Structural patterns. As explained in Section 4.1, the structural patterns of the proposed temporal property language are directly defined using VQL. Since VQL already has well defined semantics, our translation scheme simply extracts structural patterns from our temporal properties, allowing them to be executed with the semantics of VQL. At runtime, all structural patterns to be monitored are registered as observers of the execution. From then on, the VQL query engine incrementally updates the matches to its registered patterns when modifications are made to the running model. The resulting structural monitors are integrated with the temporal ones by retrieving the current matches of the former and forwarding them to the latter (see below).

Temporal patterns. Aside from a small variation on the existence pattern, the intended semantics for the temporal patterns is identical to the existing semantics of the PSPs, which was originally expressed in 3 languages, 2 of which are suitable for runtime monitoring: Linear Temporal Logic (LTL) and Quantified Regular Expressions (QRE).

However, this semantics, expressed as a mapping from pattern/scope combinations to QREs, is defined for finite state verification of infinite traces (*e.g.,* model checking), whereas in our case we seek to provide a semantics for finite execution traces to enable runtime monitoring. While there is existing work providing a semantics to LTL properties on finite traces [BLS11], the ecosystem of tools leveraging this semantics focuses on very specific approaches that do not interface well with our envisioned approach, nor with our technological space. We therefore opted for a different and novel approach: providing QRE-based mappings for the PSPs for finite execution traces, which can then be implemented using widely available CEP frameworks.

Accordingly, we adapted the existing QRE semantics of the PSPs to the needs of the approach. The result is a translation scheme compiling temporal patterns into CEP-based runtime monitors. Such monitors wait for events from the structural monitors (*e.g.,* matches for the model queries of the propositional values) and eventually deliver their final verdict to the property manager. This substantial adaptation of the original mappings of the PSPs to QREs is detailed in Section 5.

Integration using the property manager. In order to integrate structural and temporal monitors obtained using the translation scheme, we defined a *property manager* that acts as a bridge between the two kinds of monitors. Figure 6 illustrates how this works with an example execution of the activity shown in Figure 2, where `P3` is being monitored. A total of four monitors are deployed in the property manager, three of which are structural monitors evaluating the results of the `InsertCard`, `WrongPIN` and `CorrectPIN` structural patterns. The last monitor is the temporal monitor evaluating the Existence temporal pattern on the Between scope.

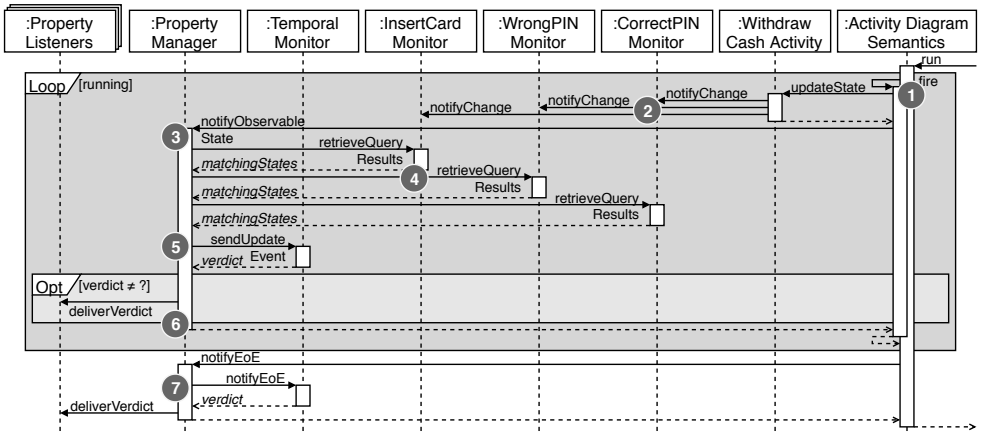


Figure 6 – Sequence diagram of a monitored execution of the *Withdraw Cash* Activity.

As the execution unfolds, the operational semantics of the Activity Diagram DSL repeatedly applies the fire execution rule, which makes changes to the model (1 on Figure 6). Changes are detected by structural monitors, causing them to update their structural pattern result (2). Once the execution rule has been applied, the execution semantics notifies the property manager that an observable state has been reached (3). This causes the property manager to retrieve the current result of each deployed structural monitor (4). From these results, the property manager sends an event containing the new values for each propositional variable of the temporal pattern to the temporal monitor. After updating its state, the temporal monitor returns its verdict for the temporal pattern it is monitoring (5). If this verdict is a final verdict —*i.e.,* it is either a violation or a satisfaction of the property— then the property manager delivers it to the registered property listeners, and the temporal and structural monitors are unplugged from the execution (6). If the end of the execution is reached without a final verdict being delivered, the execution semantics sends the corresponding notification to the property manager (7). This notification is forwarded to the temporal monitor, automatically triggering a final verdict which is then delivered to property listeners.

5 Translation Scheme of Temporal Patterns for Runtime Verification

In this section, we present the translation scheme used to derive temporal monitors from the temporal constructs of our property language.

5.1 Specificities of Runtime Verification

A core difference between our temporal property language and the PSPs is that we evaluate temporal properties at *runtime* and on *finite* executions. Therefore, a final verdict can only be rendered when an execution state that is permanently satisfying or violating the property is reached. In the context of scoped properties, such execution states come in four kinds: (1) states violating the temporal pattern of the property, (2) states satisfying the temporal pattern for the current scope in the case of single

scope activation properties (*e.g.,* Globally, Before, etc.), (3) states marking the end of the execution, and (4) states marking the end of a scope activation. Note that a single execution state can belong to the last two kinds, for instance the state marking the end of a Globally scope activation also marks the end of the execution.

While a verdict can always be rendered for the first three kinds of execution states, the fourth kind (states marking the end of a scope activation) does not guarantee this. Reaching the end of a scope activation means that no violation or satisfaction was detected on the basis of a single execution state. This in turn means that a verdict must be rendered based on the states collected by the temporal pattern over the scope activation. One of two outcomes is possible: either a definitive verdict can be rendered (property violated or satisfied), or no definitive verdict can be rendered.

In the first case, a final verdict has been reached, thus the monitoring can be stopped and observers notified of the verdict. In the second case, since the temporal pattern is evaluated independently from one scope activation to another, the resources held about the now inactive scope are not needed to render future verdicts. Therefore, we opted for a monitoring strategy at the scope scale, instead of encompassing the complete execution. As a result, we define the semantics of pattern/scope combinations to encompass individual scope activations instead of the whole execution.

5.2 Considered Target Languages

Our semantics relies on identifying matches of temporal patterns among finite sequence of states reached by an executed model, a match being a collection of execution states of interest gathered over a scope activation. Once a match has been found, it is then necessary to analyze the execution states it carries to render a verdict, *i.e.,* whether the temporal pattern is permanently satisfied (noted $\top$), violated (noted $\bot$) or no definitive verdict can be rendered yet (noted '?'). We therefore need to cover two different concerns: the expression of *what states to match*, and the expression of *verdict procedures* that analyze matches. For our approach, a temporal pattern is translated into two objects: a quantified regular expression to express what states to match, and a decision tree to express the verdict procedure.

Quantified regular expressions From a CEP perspective, matches translate well in terms of *complex events*, where the role of events is taken by propositional variables, each being defined beforehand by a structural pattern. A match can be computed as a complex event emitted upon the reception of a specific pattern of events, and that carries a set of properties whose values are computed from the received set of events.

CEP engines support multiple languages to define complex events, among which QREs. While QREs are more known for the definition of patterns of characters for string searching, they can also be used to search within a sequence of execution states. For example, given three propositional variables `Q`, `P` and `R`, `"Q ¬[P,R]* P"` will search for a match where `Q` is true in an execution state, followed directly by a possibly empty sequence of states where `P` and `R` are false, followed directly by a state where `P` is true.

As Dwyer *et al.* originally expressed the semantics of the PSPs using QREs, relying on QREs to describe our own temporal monitors makes it easier to reuse and adapt the semantics of Dwyer *et al.*.

Decision Trees Since verdict procedures are sequences of tests made on matches, we express them in this paper using decision trees. A *decision tree* is a tree where each internal node represents a test on an analyzed object, each branch represents the

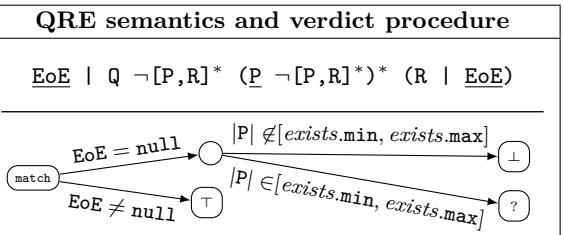
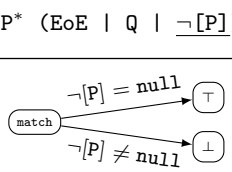
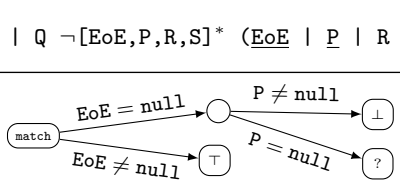
Pattern/scope combinations	QRE semantics and verdict procedure
`exists P between Q and R`	
`always P before Q`	
`S precedes P after Q until R`	

Table 2 – Excerpt of Pattern/Scope combinations and their corresponding semantics as QREs and verdict procedures.

outcome of a test, and each leaf node represents an outcome of the procedure. In our case, tests focus on specific propositional variables of interest of the corresponding QRE. We later show such variables of interest of a QRE as underlined.

5.3 A Translation Scheme for Temporal Patterns

In this part, we use one example of pattern/scope combination to explain how we adapted the QRE-based semantics of the PSPs to runtime verification. This adaptation consists both in changing the QREs proposed by Dwyer *et al.*, and on supplementing each QRE with a verdict procedure written as a decision tree.

Table 2 shows an excerpt of the translation scheme we defined for the temporal patterns of our property language, with only 3 rules out of 25 in total. Each line shows a pattern/scope combination to the left, and both the QRE and accompanying verdict procedure resulting from compilation to the right. Due to space limitation, we only focus in this part on the first row of this table, which corresponds to the Existence/Between combination. We refer the reader to the companion webpage of the paper for the complete 25 rules of the semantics[4].

5.3.1 Example of Adaptation of a QRE

Figure 7 illustrates with the Existence/Between combination how we rendered the semantics of the PSPs compatible with runtime verification. The original QRE from Dwyer *et al.* is shown at the top, then each arrow is an adaptation step.

As a first step, we restrict the QRE to a single scope activation. This is done by keeping only the part of the expression that is bound by `Q` and `R` (both included). As a second step, we ensure that the QRE is able to capture violations of the property.

[4]Companion webpage: `http://gemoc.org/ecmfa20/`

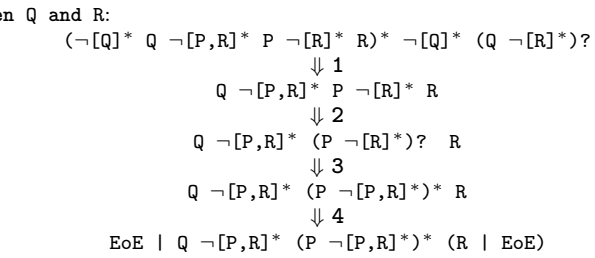


Figure 7 – Example of step-by-step adaptation of the semantics from Dwyer *et al.* [DAC98] to runtime verification.

In our example, this is achieved by making the `P ¬[R]*` pattern optional, which yields `(P ¬[R]*)?`. This way, the QRE can capture scope activations where `P` does not occur. The third step, is specific to the Existence temporal pattern: we want this pattern to gather all occurrences of `P` happening while the scope is active. This allows us to support lower- and upper-bounded Existence patterns by comparing the number of occurrences of `P` contained by a match to the `min` and `max` properties of the temporal pattern. This is achieved by changing the `(P ¬[R]*)?` sub-expression from the previous step into `(P ¬[P,R]*)*`. This way, additional occurrences of `P` that would have been captured by `¬[R]*` instead lead the `(P ¬[P,R]*)` pattern to be matched repeatedly. The fourth and final step consists in introducing an `EoE` (end of execution) event into the QRE. This event must first be added as an alternative to the whole pattern, so that when no scope is currently active, a complex event is still generated, triggering the rendering of a verdict. The second place where `EoE` must be added is as an alternative to `R`, in order to trigger the verdict rendering during a potentially active scope. To obtain our complete semantics, we repeated these adaptation steps for each of the QREs part of the semantics from Dwyer *et al.* [DAC98].

5.3.2 Example of Verdict Procedure

As we previously explained, for the purpose of runtime verification, each QRE requires a companion verdict procedure to compute an outcome from a sequence of matched execution states. A verdict procedure is executed each time its QRE finds a match, and delivers a verdict based on the properties of the complex event representing the match. These properties contain the values of a set of propositional variables of interest.

In what follows, we focus on the first row of Table 2, which corresponds to the same Existence/Between example, whose variables of interest are `P` and `EoE` (shown as underlined). The verdict procedure first checks if it was triggered by the end of the execution. If that is the case, as evidenced by the `EoE` event not being `null`, the temporal property is satisfied. This is because the Between scope does not consider the end of the execution as a valid scope upper bound, contrary to the AfterUntil scope. If the end of the execution was not reached, this means that the verdict rendition was triggered by a `R` event. In this case, the procedure counts the number of occurrences of `P` and checks that it is within the lower and upper bounds of the Existence temporal pattern. If the number of occurrences of `P` stays within these bounds, the property is neither satisfied nor violated. Otherwise, the property has been violated, and a verdict is rendered accordingly. To obtain our complete semantics, we wrote a verdict procedure for each QRE defined.

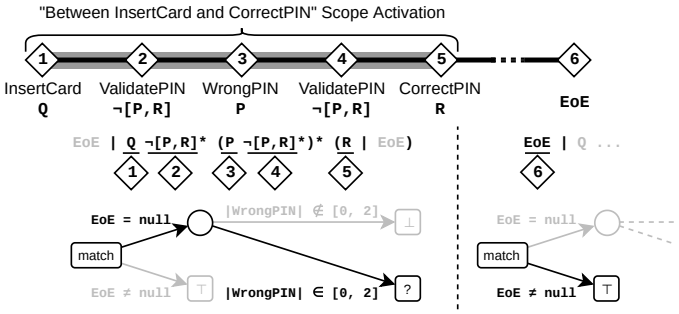


Figure 8 – Evaluation of P3 on an execution of the *Withdraw Cash* activity from Figure 2.

Example of an Execution of a Temporal Monitor. Figure 8 shows the evaluation of P3 on an example execution of the *Withdraw Cash* activity shown in Figure 2. The upper part of the figure shows an excerpt of the execution trace resulting from the execution. Here, an incorrect PIN is entered the first time, eventually followed by the correct PIN, thereby constituting a full scope activation for P3, as depicted between execution states 1 and 5 on the upper left part of Figure 8. Once execution state 5 is reached, the QRE semantics of the Existence/Between combination captured a full match, as shown on the middle left part of the figure. This match in turn triggers the associated verdict procedure, shown on the lower left part of the figure. As the end of the execution has yet to be reached and the number of WrongPIN occurrences is within the bounds of the Existence pattern, no final verdict is produced ('?').

After the verdict is rendered, the execution resumes, as shown on the upper right part of Figure 8. For this execution, no additional scope activation are encountered until the end of the execution. When the end of the execution is reached, the temporal monitor receives a corresponding notification. As the QRE of the temporal monitor is not currently matching anything, it directly captures the EoE occurrence, thereby completing a match as shown on the middle right part of the figure. This match triggers the execution of the verdict procedure, which this time returns a satisfied verdict ($\top$), as shown on the lower right part of the figure.

6 Tool Support

We implemented our approach as part of the GEMOC Studio [BDV+16], a language and modeling workbench atop the Eclipse platform [ML05]. The language workbench of the GEMOC Studio offers multiple metaprogramming approaches to define the operational semantics of a DSL (*e.g.,* Java/Kermeta [JCB+13], xMOF [MLWK13] or Henshin [ABJ+10]), as well as one execution engine for each approach. Each execution engine sends notifications when observable states are reached in the execution, which is the main prerequisite of our approach. Our implementation is thus decoupled from Kermeta and should work with other metaprogramming approaches by providing an execution engine for them.

We implemented the metamodel of the proposed temporal property language using Ecore. The structural patterns of a temporal property must be defined using the VQL language. Regarding target languages, EPL is used for QREs while nested Java conditionals are used for decision trees. Each pattern/scope combination is given a semantics as an EPL statement and a Java method to be called whenever a complex event is matched by the EPL statement. The property manager integrating the VQL

engine and the Esper engine is written in Java and Xtend. The GUI of the property manager allows to register properties to monitor and provides real-time feedback on the verdict of each registered property. The source code is open-source and accessible through the companion webpage of the paper.

7 Evaluation

In this section, we leverage the implementation described in the previous section to evaluate the execution overhead induced by monitoring temporal properties on a selection of xDSLs. Thereby, we seek to investigate how different factors influence this overhead and seek to answer the following research questions:

RQ#1 How does each property influence the execution overhead?

RQ#2 How does model size influence the execution overhead?

RQ#3 How does the footprint of properties influence the execution overhead?

RQ#4 How does execution length influence the execution overhead?

The evaluation material (models, code, data) is available on the companion web page[4].

Considered xDSLs For this evaluation, we considered three xDSLs: an extension of the *Activity Diagram* DSL presented in Section 2.1, *MiniJava* [Rob01] and *ThingML*[5]. Our *Activity Diagram* implementation was initially proposed as a solution for the Model Execution case of the Transformation Tool Contest 2015 [CDB+14]. In addition to what is shown in Figure 1, it comprises the concepts of Variable and Expression. Global variables can be declared in an activity, and the values of these variables can be changed by action nodes using integer or boolean expressions. Guards can also be used in the control flow using boolean expressions. *MiniJava* is a subset of Java created for teaching purposes. *ThingML* is a DSL for designing and implementing distributed reactive systems. It combines asynchronously communicating statecharts and components, an imperative platform-indenpendent action language and constructs targeting IoT applications. All three DSLs were implemented with the GEMOC Studio using Ecore for the abstract syntax and Kermeta for the operational semantics.

Considered Models To evaluate how the approach scales with regards to different factors, we generated models covering two factors: *model size* and *execution length*. This allows to evaluate the impact on the induced overhead of an increased search space and execution length. Models for *Activity Diagram* and *MiniJava* are generated from the same template: a loop performing calculations on integer variables for a number of iterations. For these DSLs, we modulate model size by replicating both the integer variables that are part of the structural patterns used by the evaluated temporal properties, and the calculations made on these variables. We cover models containing around 15, 150 and 1500 integer variables, referred to as factors S, M and L, respectively. To modulate execution length, we increase the number of executed iterations. We cover models iterating 10 and 100 times, thereafter referred to as *short* and *long* execution lengths, respectively. Models for *ThingML* consist of clients sending registration notices and signal to servers. When servers receive a signal from all of their registered clients, they send them a signal in response. Clients can switch to another server or shutdown depending on the total number of received signals.

[5] `https://github.com/TelluIoT/ThingML`

For *ThingML*, we modulate model size by adding more clients to the system, and we modulate execution time by changing the number of signals sent by clients. We cover models containing 5, 25 and 75 (resp. S, M, and L) each sending 3 to 20 (resp. short and long execution lengths) signals to the servers.

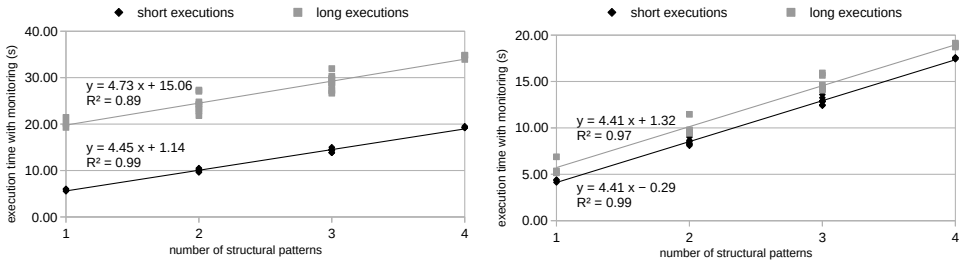
Considered Temporal Properties In order to evaluate how the approach scales with regard to *property footprint*, we generated temporal properties whose structural patterns cover portions of the models of various sizes. For *Activity Diagram* and *MiniJava*, we consider temporal properties that focus on the variables of the models. These variables are designed so that each structural pattern featured in temporal properties has its corresponding variables. For instance, the structural pattern tied to the `Existence` temporal pattern searches for a variable named `"existVar_0"`. The footprint of `Existence` properties is multiplied by 10 by generating structural patterns that require matches on `existVar_0` through `existVar_9`. For these 2 DSLs, we cover structural patterns matching 1, 10 and 100 variables, thereafter referred to as S, M and L, respectively. In the case of *ThingML*, temporal properties are based on patterns that check the current state of specific clients. For example, one type of client is expected to receive 5 events between 2 register notices, so we can check that the state `Waiting`, reached after receiving a signal, exists 5 times between the states `Registered` and `Register`. For this DSL, we cover structural patterns matching 1 (S), 5 (M) and 10 clients (L). Finally, all temporal properties are designed to be validated at the end of the execution only, to measure their overhead during the complete execution.

Experimental Protocol The execution overhead of each combination of temporal property, model size, execution time and property footprint is measured as follows. First, a warm-up phase consisting of 10 executions of the model while monitoring the property takes place. We then collect the execution time of 20 additional executions, still while monitoring the property, and compute the average execution time. We do the same for the mean base execution time of the model, without monitoring. Summaries of the resulting measurements are presented in Table 3.

Execution length	Short						Long						mean
Model size	S	M		L			S	M		L			
Property footprint	S	S	M	S	M	L	S	S	M	S	M	L	
Baseline execution time (in s) / Mean execution time (in s) / *Mean relative execution overhead* (in %)													
MiniJava	0.05	0.21		1.44			0.25	1.15		11.98			0.61
	0.11	0.27	0.31	1.77	1.84	12.03	0.34	1.87	1.89	16.22	15.72	26.98	1.95
	134.5	*23.5*	*45.0*	*21.1*	*24.9*	*725.8*	*38.7*	*63.0*	*63.7*	*34.7*	*30.1*	*122.4*	*56.5*
ThingML	0.10	0.73		5.10			0.31	2.17		15.35			1.27
	0.19	1.20	1.25	9.30	9.30	9.17	0.49	3.66	3.62	24.49	24.02	24.03	4.08
	77.1	*62.6*	*69.4*	*82.0*	*82.0*	*79.3*	*59.3*	*68.8*	*66.8*	*59.3*	*56.2*	*56.4*	*67.6*
Activity Diagram	0.03	0.05		0.08			0.17	0.19		0.60			0.12
	0.09	0.11	0.16	0.23	0.26	10.36	0.24	0.35	0.40	1.52	1.60	12.06	0.56
	168.7	*120.4*	*230.5*	*194.3*	*234.9*	*13053.6*	*40.2*	*83.2*	*105.0*	*150.9*	*164.2*	*1891.0*	*244.6*
All 3 DSLs	*120.1*						*79.6*						*97.8*

Table 3 – Average execution overhead for each factor (using geometric mean, all properties combined, S=small, M=medium, L=large).

Analysis Table 3 shows, for each combination of factors and for each DSL, the base execution times of the models, the (geometric) mean execution times while monitoring



(a) MiniJava (b) Activity Diagram

Figure 9 – Execution time per number of structural pattern in properties (model size L, property footprint L).

across all properties, and the resulting relative execution overheads.

The results show that the average overhead ranges from 56% (for *MiniJava*) to 244% (for *Activity Diagram*). A more detailed look shows that, for *Activity Diagram*, runtime monitoring on short executions induces a very high (and even extremely high, in the case of the highest property footprint) overhead: from 168% to 13053% in the most extreme case, whereas the overhead on longer executions is more reasonable on property footprint S (40% to 150%) and M (105% to 164%), but still very high on property footprint L, with 1891% execution overhead. In comparison, the overhead measured for *ThingML* stays within a reasonable range for all factors (from 56% to 82%), whereas in the case of *MiniJava*, both model sizes S and L show, respectively, high (134%) to very high (725%) overhead on the shortest execution time. Using Table 3, we answer each of the considered research questions in the following.

Answering RQ#1 From Table 3, one can observe that for both execution lengths, the execution times while monitoring properties with footprint L are between 10s and 15s higher (*i.e.,* one order of magnitude higher) than the baseline execution time for *MiniJava* and *Activity Diagram*, while staying closer to the baseline (*i.e.,* within the same order of magnitude) for footprint S and M. The same discrepancy between footprint L and S/M cannot be observed for *ThingML*.

Looking at the detailed data (available on the companion web page) reveals that the overhead induced by property footprint L for *MiniJava* and *Activity Diagram* is highly dependent on the number of structural patterns contained in the properties. This is illustrated by Figures 9a and 9b, which provide a more detailed look at these two outliers and show that there is a highly-correlated (with R^2 going from 0,89 to 0,99) relationship between the number of structural patterns contained in the properties and the resulting overhead. This relationship estimates the absolute induced overhead per structural pattern to be situated between 4,41s and 4,73s.

We thus explain the difference between the 3 DSLs as follows: there is a steep initial overhead depending on the amount and size of structural patterns on a property, that gets compensated by longer execution times. For *ThingML*, the induced overhead is reasonable because, on one hand structural patterns for our *ThingML* models are smaller by an order of magnitude and thus their initialization has less impact on the induced overhead, and on the other hand, *ThingML* is the slowest of all 3 languages, as evidenced by the baseline execution times shown in Table 3.

In the end, the most salient source of overhead from the kind of property being monitored comes from the *number of structural patterns* a property contains. This source of overhead appears to be confined to the *initialization phase* of structural patterns, and therefore is smoothed out on longer executions. Identifying the influence

of each pattern and scope making up temporal properties requires a more in-depth statistical analysis focused solely on this aspect, which we reserve for future work.

Answering RQ#2 For *MiniJava*, increases in model size greatly increase the execution time, resulting in a reduced relative overhead: without monitoring any property, going from model size S to L results in around 28 times longer executions. For *ThingML*, increases in model size have even more impact on the execution time (going from size S to L results in executions around 51 times longer), further compensating the initial overhead. Finally, for *Activity Diagram*, increases in model size do not sufficiently increase the execution time to have noteworthy compensating effects.

From these observation, and especially by comparing the respective overheads of the 3 DSLs, it appears that increase in model size has a relatively small influence on the induced overhead due to the search space increase for structural patterns, which in the case of *MiniJava* and *ThingML* is compensated by the increased execution time resulting from increased model size. In future work, we plan to further investigate if pattern search space is tied to structural pattern initialization time.

Answering RQ#3 Due to the steep initial overhead from structural patterns identified above, increasing property footprint by an order of magnitude can have a drastic effect on the induced overhead, especially on the shorter execution times. This initial influence of property footprint is compensated on longer execution time. Due to the high initial overhead induced by structural patterns, our data does not allow us to conclude on the role of increased model footprint on the overhead, past the initial one.

Answering RQ#4 Finally, the results show an average overhead of 120% on short executions, and of 79% on longer executions. Looking at the detailed data also shows that longer executions smooth out the difference of overhead between temporal properties. This again hints at an absolute overhead induced by using the approach at all, both compensated and smoothed out across the properties on longer executions.

Summary We conclude that the approach is well-suited for testing and interactive debugging, where an average execution overhead of 79% for middle length executions, and of 120% for very short executions are reasonable. While this seems high compared to existing language-specific approaches, we argue that state-of-the-art approaches rely on language-specific, low-level optimizations whose reproduction in a generic approach raises new scientific challenges. For scenarios such as live analytics, an asynchronous variant of the approach would be more suitable, and is left for future work.

8 Related Work

In this section, we explore three threads of related work: (i) works on DSLs designed to facilitate the definition of runtime monitors, (ii) runtime monitoring approaches for specific xDSLs, and (iii) approaches enabling CEP for dynamic models. To our knowledge, we provide the first generative approach applicable to a wide range of discrete-event xDSLs, and therefore compare these works qualitatively and not quantitatively to our approach.

Several works take inspiration in the PSPs to design usable languages to define runtime monitors, or to evaluate their expressiveness. Li *et al.* [LJH06] propose a language based on the PSPs to define constraints for web service interactions. A translational semantics to finite state automata is provided for each scope and temporal pattern, allowing to obtain runtime monitors from the constraints. Simmonds *et al.* [SGC+09] propose a property specification language based on Sequence Diagrams to express conversations between web services. They provide a formal semantics for their

language, allowing them to derive runtime monitors from properties defined with it. They evaluate the expressiveness of the language by using it to reproduce the PSPs.

Closer to our work, Drey *et al.* [DT16] propose a design pattern for integrating runtime monitors with the execution semantics of xDSLs. In contrast with our approach, a monitoring semantics must be provided, whereas our approach comes with a monitoring semantics of its own. Other approaches focus on specific existing DSLs. For instance, Barnawi *et al.* [BAE+15] propose a runtime monitoring approach for BPMN. In this work, the authors rely on Esper to check for violations of compliance patterns, which are derived from the PSPs. In [İA18], the authors present an approach of runtime monitoring for IoT systems. They define an event calculus describing IoT system constraints together with an event processing algebra using Esper at runtime to detect violations of the constraints of the system. While not strictly runtime monitoring, Meyers *et al.* [MDDV16] propose a generative approach to obtain testing support for an xDSL. This approach extends their previous work, proposing a generative approach to obtain, from an xDSL, a domain-specific property language inspired from the PSPs [MDL+14]. It also requires placeholder rules in the execution semantics to be replaced at runtime by calls to the testing engine.

In [DRV14, DRV18], the authors use Viatra for CEP, defining both structural and temporal patterns in an extended version of VQL comprising complex, possibly temporal patterns. While Viatra-CEP could have been our choice for the temporal aspect of the approach, we chose Esper as it is better suited for some of our planned future work, such as including, in the temporal properties, the reception or emission of specific stimuli by the model, which are not necessarily defined as model changes. In [BBTV18], the authors propose an extension of CEP in the context of graph queries, mainly by incorporating spatial windows, which allow to restrict the matching process to a subset of the whole space. However, with this approach, the definition of the structural aspect of temporal properties is done with GraphX, an API of Apache Spark, which offers a lower level of abstraction than VQL. Yet, investigating how an approach leveraging spatial windows fares, performance-wise, in comparison to incremental query evaluation is an interesting direction to take for future work.

9 Conclusion and Future Work

We have presented a generic approach for xDSLs to turn temporal properties into runtime monitors. While the approach already shows promising benefits, several future lines of research are ahead. First, various optimization techniques for reducing the runtime overhead should be investigated. Performing more in-depth analysis on the performances of the approach, specifically using statistical analysis to determine the relations between the various factors is also envisioned. Lastly, extending the approach to include the missing temporal patterns, time windows, and the support of parameterized and asynchronous monitoring are also left for future work.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *MODELS'10*. Springer, 2010.

[BAE+15] Ahmed Barnawi, Ahmed Awad, Amal Elgammal, Radwa El Shawi, Abduallah Almalaise, and Sherif Sakr. BP-MaaS: A Runtime Compliance-Monitoring System for Business Processes. In *BPM (Demos)*, 2015.

[BBTV18] Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. Extending complex event processing to graph-structured information. In *MODELS'18*, 2018.

[BCCG07] Réda Bendraou, Benoit Combemale, Xavier Crégut, and Marie Pierre Gervais. Definition of an executable SPEM 2.0. In *APSEC'07*. IEEE, 2007.

[BDH+15] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In *ICMT'15*, 2015.

[BDV+16] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool Demo). In *SLE'16*, 2016.

[BGPS12] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *ICSE'12*. IEEE, 2012.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011.

[BSVV18] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries for runtime monitoring of cyber-physical systems. In *FASE'18*. Springer, 2018.

[BSVV19] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries over models@ run. time for runtime monitoring of cyber-physical systems. *STTT*, 2019.

[BURV11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A Graph Query Language for EMF Models. In *ICMT'11*, 2011.

[CCP12] Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *APSEC'12*, 2012.

[CDB+14] Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, and Didier Vojtisek. A solution to the ttc'15 model execution case using the gemoc studio. In *8th Transformation Tool Contest*. CEUR, 2014.

[CFAI17] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A Survey of Runtime Monitoring Instrumentation Techniques. *EPTCS*, 2017.

[DAC98] Matthew B Dwyer, George S Avrunin, and James C Corbett. Property specification patterns for finite-state verification. In *FMSP'98*. ACM, 1998.

[DRV14] István Dávid, István Ráth, and Dániel Varró. Streaming model transformations by complex event processing. In *MODELS'14*. Springer, 2014.

[DRV18] István Dávid, István Ráth, and Dániel Varró. Foundations for streaming model transformations by complex event processing. *SoSyM*, 2018.

[DT16] Zoé Drey and Ciprian Teodorov. Object-oriented design pattern for DSL program monitoring. In *SLE'16*. ACM, 2016.

[EHHS00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *UML'00*. Springer, 2000.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnuelli, Michal Politi, Rivi Sherman, Aharon Shtull-trauring, and Mark Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *TSE*, 1990.

[İA18] Koray İnçki and Ismail Ari. A novel runtime verification solution for IoT systems. *IEEE Access*, 2018.

[JCB+13] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the Kermeta language workbench. *SoSyM*, 2013.

[LJH06] Zheng Li, Yan Jin, and Jun Han. A runtime monitoring and validation framework for web service interactions. In *Australian Software Engineering Conference (ASWEC'06)*. IEEE, 2006.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 2009.

[MDDV16] Bart Meyers, Joachim Denil, István Dávid, and Hans Vangheluwe. Automated testing support for reactive domain-specific modelling languages. In *SLE'16*. ACM, 2016.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Promobox: a framework for generating domain-specific property languages. In *SLE'14*. Springer, 2014.

[ML05] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.

[MLWK13] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *SLE'13*. Springer, 2013.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. URL: `https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[Obj13a] Object Management Group. OMG Unified Modeling Language (OMG UML), V 2.5, September 2013. `http://www.omg.org/spec/UML/2.5`.

[Obj13b] Object Management Group. Semantics of a Foundational Subset for Executable UML Models, V 1.1, August 2013. `https://www.omg.org/spec/FUML/1.1`.

[Obj16] Object Management Group. Meta Object Facility (MOF) Core Specification, V 2.5, June 2016. `http://www.omg.org/spec/MOF/2.5`.

[Rob01] Eric Roberts. An overview of MiniJava. *ACM SIGCSE Bulletin*, 2001.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Eclipse Series. Addison-Wesley Professional, 2008.

[SGC+09] Jocelyn Simmonds, Yuan Gan, Marsha Chechik, Shiva Nejati, Bill O'Farrell, Elena Litani, and Julie Waterhouse. Runtime monitoring of web service conversations. *TSC*, 2009.

[TCGT14] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *MODELS*. Springer, 2014.

About the authors

Dorian Leroy (First Author) is a PhD student doing an international PhD between the JKU Linz (Austria) and the University of Rennes 1 (France), currently in the DiverSE team in Rennes. His research interests lie in the field of Software Language Engineering and include metaprogramming approaches and generic V&V facilities. Contact him at `dorian.leroy@jku.at`, or visit `https://d-leroy.github.io/`.

Pierre Jeanjean is a PhD student at Inria (France), currently working in the DiverSE team in Rennes. His research interests include Software Language Engineering and Domain-Specific Languages, and more specifically interactive language interpreters (REPLs) and IDE features. Contact him at `pierre.jeanjean@inria.fr`.

Erwan Bousse is an Associate Professor at the University of Nantes (France). He obtained his PhD in France in 2015 at the University of Rennes 1 for his work on execution traces and omniscient debugging of executable models. His current research interests include Software Language Engineering (SLE), Model Driven Engineering (MDE), Domain-Specific Languages (DSLs), model execution and simulation, and the debugging and testing of models. Contact him at `erwan.bousse@ls2n.fr`, or visit `https://bousse-e.univ-nantes.io/`.

Manuel Wimmer is Full Professor leading the Institute of Business Informatics - Software Engineering at the Johannes Kepler University Linz and he is the head of the Christian Doppler Laboratory CDL-MINT. His research interests comprise foundations of model engineering techniques as well as their application in domains such as tool interoperability, legacy modeling tool modernization, model versioning and evolution, and industrial engineering. Contact him at `wimmer@jku.at`, or visit `https://www.se.jku.at/manuel-wimmer/`.

Benoit Combemale is Full Professor of Software Engineering at the University of Toulouse, and a Research Scientist at Inria. His research interests are in the field of software engineering, including Model-Driven Engineering, Software Language Engineering and Validation & Verification; mostly in the context of (smart) Cyber-Physical Systems and Internet of Things. He is also teaching worldwide in various engineering schools and universities. Contact him at `benoit.combemale@irisa.fr`, or visit `http://combemale.fr`.

Acknowledgments This work has been partially supported by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development (CDG), and by the FWF under the grant numbers P28519-N31 and P30525-N31.